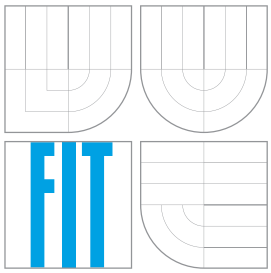
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ZVÝŠENÍ VÝKONU NITRATE KLIENTA POMOCÍ VYROVNÁVACÍ PAMĚTI

NITRATE CLIENT PERFORMANCE IMPROVEMENT WITH CACHE IMPLEMENTATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — FILIP HOLEC
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR MÜLLER
SUPERVISOR

BRNO 2013

Abstrakt

Cílem práce je návrh a implementace výkonnostních vylepšení modulu python-nitrate. Výkonnostní vylepšení jsou založeny na sesbíraných případech užití, které využívají velké množství dat. Za účelem měření dopadu změn v modulu byly implementovány výkonnostní testy. Testování ukázalo, že modul python-nitrate s integrací vylepšení je v některých případech až několikanásobně rychlejší, avšak ve dvou případech může nastat zpomalení. Závěr práce obsahuje diskusi ohledem pokračování prací.

Abstract

The goal of the thesis is to design and implement performance improvements in python-nitrate module. Performance improvements are based on gathered use cases, which use large amount of data and network bandwidth. Performance test suite was implemented in order to measure impact of changes in module. Testing proved, that python-nitrate module with integrated performance improvements is in certain cases several times faster, but also can be slower in two cases. Discussion regarding possible extensions is present in the conclusion

Klíčová slova

Výkonnostní testování, vylepšení kešování, Nitrate, Python, python-nitrate, MultiCall, XML–RPC

Keywords

Performance testing, cache improvement, Nitrate, Python, python-nitrate, MultiCall, XML–RPC

Citace

Filip Holec: Nitrate Client Performance Improvement with Cache Implementation, bakalářská práce, Brno, FIT VUT v Brně, 2013

Nitrate Client Performance Improvement with Cache Implementation

Declaration

I declare this thesis is my work and it has been created under supervision of Petr Müller and Petr Šplíchal from Red Hat. Every source has been correctly cited along with reference to the corresponding sources.

.....
Filip Holec
May 15, 2013

Acknowledgements

I would like to thank my Red Hat supervisor Petr Šplíchal for lead, consultations, supervision, help and valuable feedback to this thesis and Petr Müller for providing hints, consultations, feedback and willingness to help. I would also like to thank my family and friends for encouragement and support.

Contents

Chapter 1

Introduction

The goal of this bachelor thesis is to improve performance in python-nitrate module, a Python interface to the Nitrate test case management system, and create performance test suite for this module. Issues related to performance are very common and these are mainly visible when python-nitrate has to fetch hundreds and even thousands of objects. I think one of the solutions can be improvement of caching in classes, because the current one does not cover all classes. With the purpose of measuring the improvements, a new performance test suite has to be implemented. Next step is to find out what this module is currently doing and try to implement features that are not yet present. In the end, improvements are measured with the performance test suite.

This thesis is divided into two main parts, and that is the theoretical part, and part that includes performance test suite, implementation of feature enhancements and testing.

Theoretical part consists of two major chapters: detailed overview of Nitrate and python-nitrate module and performance testing. The core in this part is to understand the whole concept of python-nitrate, but also to get familiar with ways of performance testing of an application and what needs to be done to perform successful performance testing.

In the performance test suite part use cases are presented along with their test cases. This test suite is later used in testing part, where they are run under multiple variations of conditions. Improvements, testing and results of the thesis describe the process of testing improved implementations and comparison to the original implementation of the python-nitrate module. The evaluation of testing is present along with commentary.

The last chapter, conclusion, is dedicated to final assessment of the whole thesis and its contribution to python-nitrate users. Also possible future enhancements are discussed there.

Chapter 2

Nitrate and python-nitrate

2.1 Nitrate

Nitrate [4] is a test case management system (TCMS) and it's written in Python and uses Django web framework.

According to documentation, this management system provides several features:

- XML–RPC interface
- audit traceability
- increased productivity – identification of gaps in product coverage
- reproducibility across planning, cases and execution
- multiple authentication backends

XML-RPC [10] is a Remote Procedure Call method. It uses HTTP as a transport and passes XML. Using this, a client can call methods with specified parameters on a remote server (which is named by URI) and get back structured data. xmlrpclib handles all the details that are necessary to translate between conformable Python objects and XML on the wire.

Nitrate is an open source project and its source code can be viewed via browser at:

```
https://fedorahosted.org/nitrate/
```

or using git:

```
git clone git://git.fedorahosted.org/nitrate.git
```

2.2 python-nitrate

python-nitrate [9] is a Python [7] interface to the Nitrate test case management system. The package consists of a low-level driver that allows direct access to Nitrate's XML–RPC

API. In addition, python-nitrate is a high-level Python module with natural object interface and a command line interpreter, which is useful for fast debugging and experimenting.

python-nitrate fetches data from Nitrate database and interprets it, user can modify the data and update it in Nitrate instance or use it to display requested information.

The core of the thesis is to analyze python-nitrate module as a whole, suggest improvements in implementation and integrate these changes to the module. As mentioned before, improvements are going to be measured by performance test suite, that will be integrated into python-nitrate. Since many users suffer from decline in performance of specific use cases (because of redundant or time-consuming operations), these enhancements of the module are essential.

python-nitrate is an open source project with accessible source codes and RPMs that can be viewed and downloaded here:

`http://psss.fedorapeople.org/python-nitrate/download/`

2.2.1 Caching in python-nitrate

python-nitrate client supports currently 4 types of caching. First type is **CACHE_NONE**, where caching is disabled and every object change is immediately written to server. Every query is sent to server even if several same queries entered in a row. Second type is **CACHE_CHANGES**. When using this caching type, caching is enabled and changes are pushed to the server only if operation *update()* is entered or upon destruction. Fetching from server is the same as in CACHE_NONE. Third type is **CACHE_OBJECTS**, where any loaded object is saved for possible future use. Finally, caching type **CACHE_ALL** caches all available objects (those which are in class, for example if user name is required with ID 123, every user will be cached).

	Caching type			
Reference	**NONE**	**CHANGES**	**OBJECTS**	**ALL**
first	fetch	fetch	fetch and store	fetch all and store
second	fetch	fetch	use cached	use cached

Table 2.1: Caching objects

2.2.2 Classes in python-nitrate

There are several classes in python-nitrate module. Their tree structure is displayed in figure 2.1 :

Config class represents the extraction of python-nitrate configuration from user configuration file (located in `$HOME/.nitrate`). These preferences are later used (for example, the url of server, configuration of test suite, ...). Configuration is parsed using `ConfigParser` [11] module. More information are present in python-nitrate documentation.

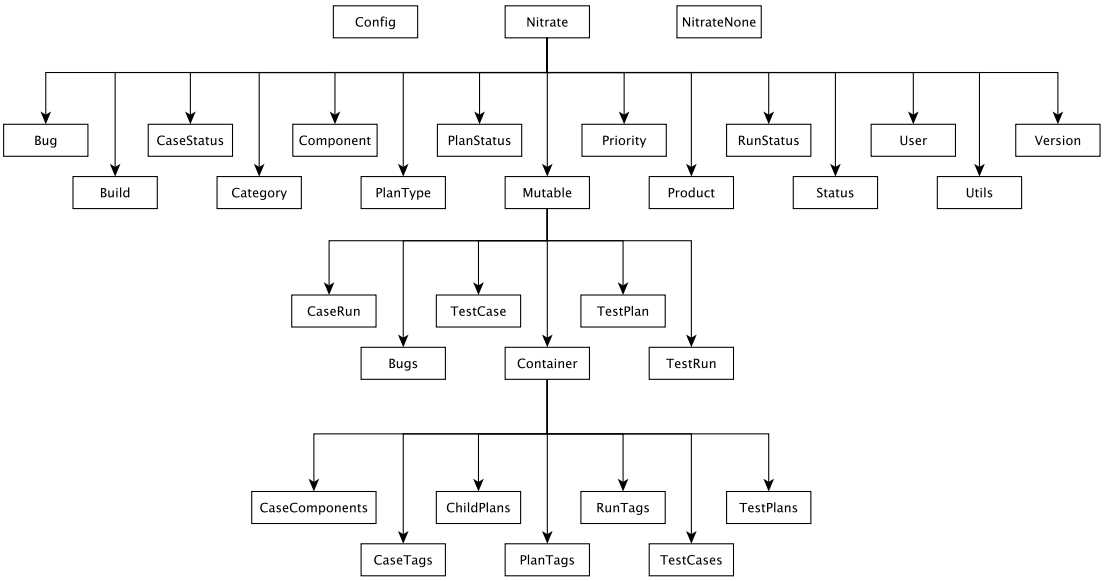


Figure 2.1: Classes overview in python-nitrate

NitrateNone is used to distinguish uninitialized values from regular "None". All attributes of an Nitrate object are initialized to NitrateNone in *__init__()*.

Nitrate is the core class and parent to every class handling objects from server. In order to develop common caching for all Nitrate subclasses, method `__new__()` in this class has to be implemented.

Bug is a class responsible for interpreting bug connected to test case and case run.

Class **Build** is connected to Product class and an object of this class contains information about build of product.

CaseStatus contains the status of specified test case. This class does not need caching, since it does not communicate with server (all case statuses are present in this class). These statuses are: PROPOSED, CONFIRMED, DISABLED, NEED_UPDATE.

Category is connected to test case and it determines the type (category) of test case, for example "Regression" test case. This class has caching CACHE_OBJECTS implemented.

An object of **Component** class specifies the component of test case. For example: wget, bash, rpm, ... Like *Category*, this class also has CACHE_OBJECTS caching.

PlanType object contains type of test plan (for example Integration, Unit, ...). This class was recently implemented along with basic type of caching (CACHE_CHANGES).

PlanStatus class represents the status of test plan. Since it can only have two states (DISABLED, ENABLED), no communication with server is required to interpret what

state is test plan in.

Class **Priority** is connected to test case and determines its priority. This class does not fetch information from server since priorities (P1 ... P5) are present there.

Objects of **Product** contain the product information. It is connected to all major Mutable classes, so caching of this class is really important.

RunStatus, like *CaseStatus* and *PlanStatus* is a class that objects does not communicate with server (only two statuses, RUNNING and FINISHED).

Class **Status** has total of 8 pre-defined statuses (thus no network communication). These states relate to case runs and are the following: IDLE, PASSED, FAILED, RUNNING, PAUSED, BLOCKED, ERROR and WAIVED

User class objects encapsulate information about user. This is the only class with CACHE_ALL caching type.

Utils only consists of tests for utility functions.

Version is connected to Product. Instances of this class hold information about versions.

Mutable is a general class for all mutable Nitrate objects. The difference between mutable and immutable objects is in the existence of *update()* method, which pushes any changes to the Nitrate server (and *_update()* method performs the actual update). Every class has its own implementation of *_update()* method.

Bugs objects embody relevant bug list for test case and case run objects. This class is not a part of cache implementation (caching is not desired as the whole class will be rewritten).

CaseRun class instances handle case run information. It is connected to test case, since case run is a test case in a test run (one test case can have multiple case runs in multiple test runs). It is uncertain whether this class will have its own cache due to lack of information stability (changing status might be quite common).

TestCase instances handle test cases that are present in test plans and it makes sense to implement caching in this class.

TestPlan objects represent test plan in Nitrate. It may contain several test cases and test runs.

TestRun is a class that handles test run instances. They contain case runs that are created from test cases in a test plan, one test run is connected to one specific test plan.

Container is a general container class for handling sets of objects. It provides the *add()* and *remove()* methods for adding and removing objects and the internal *_add()* and

_remove() which perform the actual update to the server (implemented by respective class).

CaseComponents is a container class that deals with components linked to a certain test case.

CaseTags, **RunTags** and **PlanTags** container classes process tags connected to cases, runs or plans. All tags are fetched in a single call.

ChildPlans container is used to fetch children test plans of a specified test plan in one call.

TestCases container fetches all test cases of a test plan in a single call.

TestPlans container fetches all test plans connected to a test case in a single call.

Table 2.2 shows current type of caching in classes:

	Caching type			
Class	**NONE**	**CHANGES**	**OBJECTS**	**ALL**
Bug	X	X	X	X
Build	OK	N/A	X	X
CaseStatus	N/A	N/A	N/A	N/A
Category	OK	N/A	OK	X
CaseRun	OK	OK	X	X
TestCase	OK	OK	X	X
TestPlan	OK	OK	X	X
TestRun	OK	OK	X	X
PlanStatus	N/A	N/A	N/A	N/A
PlanType	OK	N/A	OK	X
Priority	N/A	N/A	N/A	N/A
Product	OK	N/A	X	X
RunStatus	N/A	N/A	N/A	N/A
Status	N/A	N/A	N/A	N/A
User	OK	OK	OK	OK
Version	OK	N/A	X	X

Table 2.2: Classes (excluding Container subclasses) and their current type of caching

Chapter 3

Performance of an application

Why is performance testing [8] important and why do it in the first place? Badly performing applications are not a great benefit to a company or an organization. These applications mostly create a net cost of time and, of course, money. Moreover, if application does not deliver benefits, its future is not bright.

Performance testing is very significant, but underrated part of testing (unlike unit or functional testing). Sadly, it is not appreciated among executives and its importance is, as already mentioned, ignored. This fact has slightly changed in the last decade despite efforts of many known software consultants and highly publicized failures of key software applications.

Performance of application depends on perception. A well-performing application lets the user carry out a given task without irritation or perceived delay. Performant application does not display blank screen during login and can complete user's task without letting their attention wander. However, delivering acceptable level of performance is a struggle for a lot of applications.

In this context, application is being referred to as a whole, since it is composed of many parts. The higher level consists of the application software and the application landscape. For example servers required to run and also the network infrastructure for communication are latter. Either way, if problems occur in any of these areas, application performance is sure to decline.

Some people may say the best approach to ensuring good application performance is to observe the behavior of each of these areas (under load and stress) and solve any problem that occurs. This is a common mistake. since you end up dealing with the symptoms of performance rather than dealing with the cause.

3.1 Measuring the performance

There are several key indicators that must be taken into account. They could be divided into two groups: service-oriented and efficiency-oriented. **Service-oriented** indicators measure how well (or not) an application is providing a service to the users. These indicators are

availability and response time. **Efficiency-oriented** indicators measure how well (or not) an application makes use of the application landscape. These indicators are throughput and utilization.

These terms could be defined as following:

Availability is the amount of time of application's availability to the user. This is very crucial, since lack of availability could have substantial business cost even for a small outage. In other words, this would mean complete inability for user to make effective use of the application.

Response time is amount of time that takes the application to respond to user request. Measuring system response time, the time between users request for response and from the application and a complete reply to the user, is sufficient for performance testing.

Throughput is the rate at which certain application-oriented events occur. For example the number of hits on a web page in a given period of time.

Utilization is the percentage of the theoretical capacity of a resource that is being used. For example the amount of memory used when certain number of visitors is present on a server or how much bandwidth is being consumed by traffic of application.

3.1.1 Performance standards

Although there have been attempts to define a standard (particularly for web-based applications), there is no generic standard to evaluate if performance of application is good or bad. Good example is the minimum page refresh time. It came from 20 seconds rapidly to 8 seconds, but the application user wants instant response, which is likely to remain elusive.

There was a research [5] in 1986 that attempted to map user productivity to response time. Even though the research was based on green-screen text applications, its conclusions are probably still relevant.

Conversational interaction should not be greater than 15 seconds. Otherwise waiting for the answer becomes intolerable for a busy call-center operator of futures trader. The system should be designed to allow the user turn to other activities if such situation occurs.

Delays greater than 4 seconds are too long for the user to retain information in short-term memory. These could inhibit problem-solving activity and data entry. But after the transaction is completed, delays between $4 - 15$ seconds are acceptable.

For operations demanding high level of concentration, delays should not exceed 2 seconds. Waiting between 2 to 4 seconds can seem surprisingly long when the user concentrates and is emotionally commited to completing the task at hand. Response time should be than 2 seconds when the application user has to remember information through several responses. With the level of detailed information rises the need for responses shorter than 2 seconds. When work is thought-intensive (for example writing a book), in order to maintain the

users' interest very short response time is required (under 1 second). Response time to pressing a key (and displaying it) should not be greater than 0.1 seconds, as well as clicking on object with mouse. Moreover, applications like computer games require extremely fast interaction. To sum up, critical response time seems to be 2 seconds. If the delays are greater, user's productivity will almost certainly decline. Obviously, the nominal page refresh time is not ideal since 8 seconds is highly above the critical barrier.

3.1.2 Bad performance: why is it common

It is a common habit that performance problems surface late in the application life cycle. There is a rule: the later you discover them, the greater will be the cost to resolve (as you can see on valve curve).

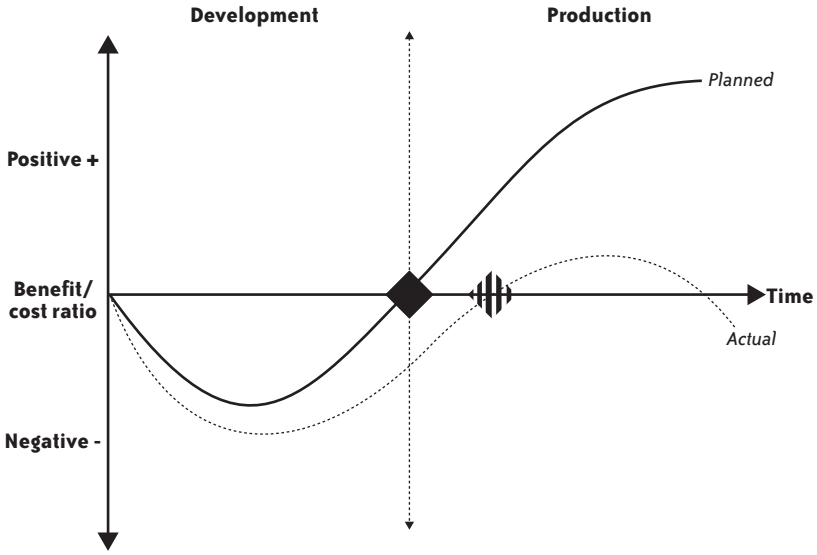


Figure 3.1: The IT Business valve curve. Source [8]

In this figure expected outcome is represented by solid line and planned moment of deployment by black diamond. If the application is released on schedule and immediately starts to provide benefit to the business with nearly no problems after release. The dotted line represents reality when development and deployment target slip, which is portrayed as striped diamond and it involves great effort to fix performance issues in production costing a lot of time and money.

3.1.3 Opinion of analysts

Forrester Research (http://www.forrester.com/) in 2006 provided interesting results by looking at the number of performance defects that had to be fixed in production for a typical application deployment. The results could be viewed in this table:

Approach	% resolved in production
Firefighting	100%
Performance Validation	30%
Performance Driven	5%

Table 3.1: Resolving performance defects

Three levels of performance were identified. The first level is firefighting, which is typical for applications where nearly no or just little performance testing was carried out in application deployment. This means, that all performance problems have to be resolved in live environment (after the application is released). Although this approach is the least desirable, it is quite common and puts companies into serious risk. The second level is performance validation and it covers companies that spent time by performance testing only in the late phase in the application life cycle. Despite this fact, still rather large number of performance defects (30%) are present in production. This level is the most common approach for organizations. Finally, performance driven approach is a stage where performance testing has been conducted at every stage of the application life cycle. The outcome of this is that only small percent of performance problems are discovered after deployment (only 5%). This should be aim for every company to adopt it as a good habit.

Last minute performance testing

Even though performance validation mode is much better than firefighting mode, last minute performance testing is still pretty dangerous. There is always a risk that serious performance defects that surface in production and the time to correct problems identified before release won't be sufficient. This could cause delays in application rollout, or application is deployed with severe performance defects and will cost a lot of time and money. Moreover, application might be withdrawn from circulation until problems are fixed. These outcomes affect negatively the business and the confidence of the users of the application. The best approach is to test for performance as soon as possible instead than postpone it to last minute.

Another unfortunate habit is that developers and testers overlook the amount of people in user community and their geography (developers are likely to ignore large number of users who have low-bandwidth, high-latency WAN links. Underestimating the popularity of (mostly) web application is no strange thing either. For example, if you estimate 10000 hits on your new web page and it suddenly becomes 1 million hits, your application infrastructure will most likely collapse.

3.2 Fundamentals of effective application performance testing

Performance awareness should be built into the application life cycle as early as possible

Performance requirements can be divided into functional an nonfunctional requirements.

Regression and unit testing are considered as functional, performance related requirements can be considered as nonfunctional requirements. These are very important when trying to carry out effective performance testing.

Performance testing is much more than generating load and seeing what happens. Many other factors must be taken into account before appropriate performance testing strategy can be implemented. According to book The Art of Performance Testing [8], these are the most important requirements for performance testing:

- choose appropriate performance testing tool
- design appropriate environment for performance testing
- setting performance targets that are realistic and appropriate
- application has to be stable for performance testing
- obtaining a code freeze
- scripting and identifying critical transactions for business
- high quality test data
- accurate performance test design
- identification of server and network monitoring key Performance Indicators (KPIs)
- enough time to performance test effectively

Although a lot of these requirements are obvious, some of them are not. Generally, it's the requirements you underestimate have actually the greatest impact on the success or failure of performance testing. Every point will be examined in detail.

3.2.1 Choosing appropriate performance testing tool

During the last 15 years, automated tools transformed from „fat-client“ norm to web-enablement. This norm is better for performance tester because of more automated tools vendors to choose from, thus tester can choose from offering with even low budget (open source tools are available as well on http://www.opensource.org/). However, if the needs of performance testing move outside the Web, the number of available tools decreases and obsolete (and bad) technologies are still present. These technologies center on recording application activity and modifying resulting scripts for performance test rather than execution and analysis. Web-based technologies can cause some problems for performance testing tools too. For example, not all tools will be able to offer a solution when dealing with streaming media or client certificates. Despite these problems, automated tools are required in order to carry out serious performance testing. There is no practical way to perform reliable (and repeatable) performance testing without some form of automation.

The main goal of automated performance testing tools is to simplify the testing process. Normally, automated tools record user activity and render this data as transactions

or scripts. These scripts are later used to create scenarios that represent a mix of typical user activity or to create load testing sessions and can be considered as actual performance tests. Once created, they can easily be reused, which is a great advantage compared to manual testing. One of the greatest advantages over manual testing is the option to correlate performance data from various sources (for example the network, servers, application response time) and display them in a single view. This information is stored for each test run, thus making comparison of the multiple results easy.

3.2.2 Testing tool architecture

Typical components of an automated performance test tools are:

Scripting module enables recording of user activity and possibly support many middleware protocols. Modification of the scripts should be allowed to associate internal and external data and to configure granularity of response-time measurement.

Test management module is responsible for the creation and execution of load tests sessions or, for example, scenarios that represent different mixes of user activity. The sessions use scripts and one or more *load injectors* (which generate the load, it could generate from multiple workstations or servers, depending on the amount of load that is required).

Analysis module has the ability to analyze the data collected after test is executed. Obtained data is generally a mixture of autogenerated reports and the report is in graphical or tabular form. Also, an 'expert' capability could be present (automated analysis of results and point out areas of concern).

Monitoring network and server performance while load is test is running is possible with additional modules. In Figure 2 is a demonstration of a typical performance tool deployment. Group of servers or workstations will inject application load and will represent application users by creating 'virtual users'.

3.2.3 Appropriate performance test environment

Ideally, the best test environment would be an *exact copy* of the deployment environment, but this case is very rare (for a number of reasons). Therefore the typical performance test environment could be a *subset of the deployment environment.* But you should certainly attempt to make the performance test environment as close to a replica of the live environment as possible within existing constraints. This is different from unit testing, where the goal is to ensure that the application works correctly.

It could take several weeks or even months to set up acceptable performance test environment, since the process is rather difficult. Therefore, to complete this activity, you need to plan for a realistic amount of time. Understanding the entire test environment enables more efficient test design and planning. Moreover, it can help to identify testing challenges early in the project. Occasionally, this process must be revisited periodically throughout the life cycle of the project.

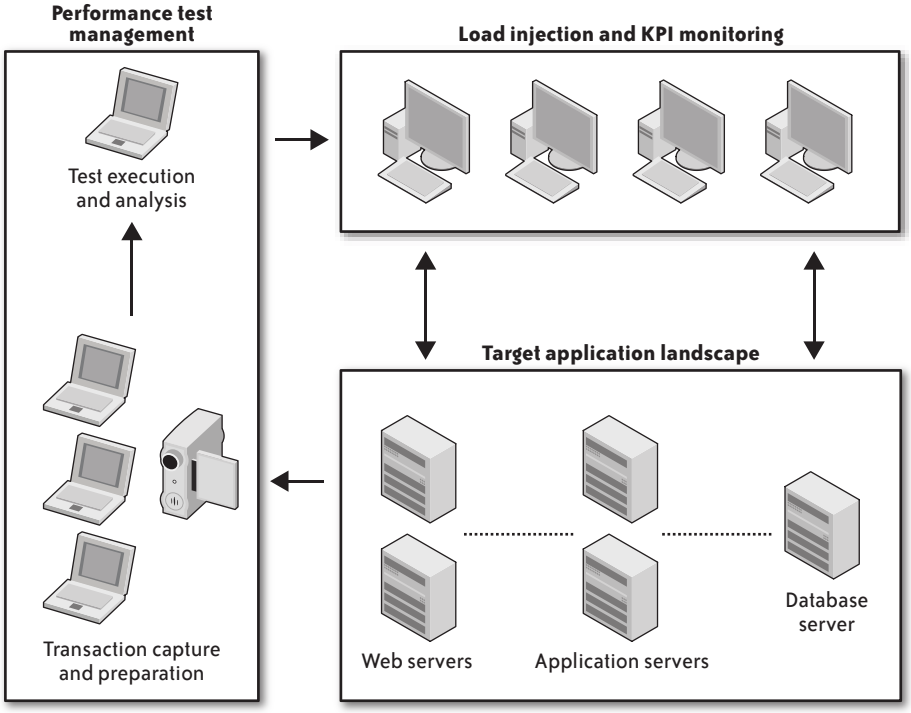


Figure 3.2: Performance tool deployment. Source [8]

Virtualization

Virtualization is a relatively new factor influencing design of test environment, which allows multiple 'virtual' servers to exist on a single physical machine. Ideal conditions are when live environment also makes use of virtualization. If so, very close approximation will be possible between live and test environment.

Injection Capacity

In order to generate the required load for performance testing, you need to ensure that hardware resources are sufficient. One or more machines are used to simulate real user and to generate load using automated performance test tools. Of course, there is a limited number of users you can generate from machine (it depends on technology). Another important task is to make sure that none of the injectors are overloaded (CPU or memory utilization), because it may have severe impact on results of the performance test. Representing many users with only one machine is a compromise in automated performance testing, so the goal should be to use as many machines to represent load injection as possible (and thus spreading the load).

3.2.4 Realistic performance targets

Performance targets are often referred to as performance goals or a Service Level Agreement (SLA). It is crucial to have clearly defined performance testing targets, otherwise it may be a waste of time.

Consesus

Consensus on the performance targets is critical, so everyone involved and on whom the application will have an impact, from application users to senior management must agree on the same performance targets.

Unfortunately, promoting consensus have not been important, since performance testing has been always last-minute activity or even completely omitted. Gaining consensus on performance testing projects within an organization should contain promoting a culture of consultation and involvement. Therefore, interested parties should be involved in the project at its early stage.

The following groups or individuals should be ideally involved:

- Departmental Heads
- The developers and testers
- The infrastructure team
- The application users

Key Performance Targets

Generally, there are three performance targets that apply for any performance testing. These are based on service-oriented performance indicators:

- Availability or uptime
- Concurrency, scalability and throughput
- Response time

The following, which are as much a measure of capacity as of performance, can be added:

- Network utilization
- Server utilization

Availability or uptime

This is very simple requirement: the application has to be available to the user at all times, the only exception is planned maintenance. It certainly must not fail within the target level of throughput or concurrency. Testing the availability is not as simple as it seems. For example, a successful ping of the server's physical machine does not necessarily mean the application is available. Another important parameter here is load. The application might run very well at modest loads, but when load increases, it may start to time out and return errors, thus suggesting lack of capacity for the generated load.

Concurrency, scalability and throughput

First, the definition of the word *concurrency* is required. According to Scott Barber's white paper [2], concurrency is (from the perspective of a performance testing tool) the number of active users generated by the software, which is not necessarily the same as the number of users accessing the application concurrently. The point is, that capacity goals can be derived from concurrency and scalability. Achieving the scalability targets demonstrates sufficient capacity in the application landscape for the application to deliver to the business.

In terms of performance testing, two distinct areas are referred as *concurrency*:

Concurrent virtual users are understood as the number of active virtual users. This number is often different from the number of users actually accessing the testing application.

On the other hand, *concurrent application users* is the number of users that are currently accessing (are logged in) the tested application. This is key measure of how many virtual users are active in a certain moment. Another thing that needs to be decided is if the process of login and logout (and everything it involves) will also be a part of application activity testing. If we decide to include it to include these processes, users which are logged out will be not truly concurrent with other users. There are a lot of solutions to this problem (increase the execution time or persistence of each transaction, ...), but if these processes are not included as a part of testing, it becomes a whole lot easier.

The 80/20 rule applies (among many other things) also in performance testing: out of 100 users of application, around 20 users will be using the application anytime during the working day. Of course, allowances for usage peaks outside of normal limits should be included in testing. For example, in a large university, day-to-day usage could be relatively flat, but only at certain times of the year (student enrollment, published results, registration of projects, ...) concurrent usage increases significantly.

When the application is considered as stateless, the performance testing target is *throughput* rather than concurrency. This situation happens, when the application has no concept of the traditional logged-in user. This kind of performance is measured as 'hits' per minute or per second.

Response Time

As mentioned before, good response time is a matter of perception.

Network Utilization

The impact of network utilization on performance testing depends on the available bandwidth between servers and the user. Exhausting available bandwidth is much less probable in a modern data center compared to in-house testing. However, when moving closer to the user side, change in performance can be significant - especially when communication involves the Internet. Large numbers of network conversations with high data presentations rates will have strongest impact on the transmission path with the lowest bandwidth.

Typical network metrics that should be measured while performance testing include the following:

Data volume is the amount of data presented to the network. This is fairly important when users will be connecting to the application over low-bandwidth WAN links. High data volume combined with network latency effects and bandwidth restrictions does not usually yield good performance.

Data throughput is the rate that data is presented to the network. Performance target could be just several bytes per second. This target can be achieved by monitoring data throughput and it can discover if any problems are occurring. Unexpected reduction in data throughput is often the first indication of capacity problems, where the servers cannot keep up with the requests and virtual users start to suffer from time-outs.

Data error rate occurs when large number of network errors that require retransmission of data slow down throughput, thus degrading performance of the application.

Server utilization

Server resources that an application is allowed to use might be limited. This can be determined by monitoring KPIs while the server is under load. Many server performance metrics can be monitored, but the most important are:

- CPU utilization
- Memory utilization
- Disk I/O (input and output)
- Disk space

3.2.5 Stable application for performance testing

When the test environment is provided and performance targets are set, application stability for performance testing has to be confirmed. This may seem obvious, but performance testing turns quite often to bug-fixing exercise and time for testing declines rapidly.

Stability of an application could be defined as the confidence that an application does exactly what it says on the box. If there are serious problems with the functionality of the application, there is no point in performance testing, since these problems will mask important problems that are results of stress and load. Naturally, code quality is significant factor in performance testing and is paramount to good performance.

There are several areas that can hide problems:

High data presentation might be a serious problem even though the application is functionally stable and it can be due to coding or design inefficiencies. User restrictions in bandwidth will certainly have negative impact on performance. Therefore application should not have redundant conversations between client and server, also excessive data (large images, ...)

within a web page are not preferred.

Poorly performing SQL is another problem, where if using SQL database, bad coding or configuration of stored procedures usually causes decline in application's performance. These flaws have to be identified and corrected, since this effect on the performance testing will only be magnified under increasing load.

Large number of application network round trips would make application vulnerable to the effects of latency, bandwidth restriction and network congestion.

Undetected application errors may be also a problem. Even though the application works properly from a functional perspective, there might be errors that are not apparent to the users or the developers. They may be creating inefficiencies that can impact performance and scalability. For example, several HTTP 404 Not Found errors in a single transaction might not be a big problem, but multiply it by several thousands transactions per minute could have serious impact on performance.

3.2.6 Ensuring Accurate Performance Test Design

When key transactions and their data requirements are identified, the next step is creating number of different types of performance tests. There are several types of tests:

Baseline test is used to establish a point of comparison for further test runs (mainly for measuring response time of transaction). This test is executed as a single virtual user for a single transaction for a set period of time (or for set number of transaction iterations). There should be no other activity on the system involved, since the goal is to provide 'best case' measurement. Obtained value is used to determine the amount of performance degradation, in response to increasing number of users or throughput.

Load test is a standard performance test, in which the application is loaded up to the target concurrency. It is the closest approximation of real application use, including simulation of user interaction with the application client. Delays and pauses experienced during data entry are taken into account as well as responses to information returned from the application servers.

Stress test causes the application or some part of infrastructure to fail in order to determine the upper limits or sizing of the infrastructure. Stress test continues until something breaks: response time exceeds the value specified as acceptable, no more user can log in or the application becomes unavailable. The point of this testing is when for example the target concurrency is 2000 users and the infrastructure fails at only 2005 users, it is good to know this because it shows that there is very little extra capacity available.

Soak or stability testing identifies problems that may appear only after extended period of time. The most common example is a slowly developing memory leak or an unpredicted limitation in the number of times that a transaction can be executed. Server monitoring is required when carrying out these type of tests. The problems will manifest typically as a gradual shutdown in response time or as an application's sudden loss of availability. In order to ensure accurate diagnosis, the correlation of data from users and servers at the

point of failure is vital.

Smoke test focuses on what has changed. Therefore, smoke test should involve only those transactions that have been affected by a code change.

Isolation test usually consists of repeated executions of specific transactions that have been identified as a result of a performance issue.

Baseline, load and stress test should be always executed. Smoke test and soak test are more dependent on application and the time

3.3 Server and Network KPIs (Key Performance Indicators)

Identifying and monitoring server and network performance metrics should be done for the application. This is vital to achieve root-cause analysis of any problems that can appear while performance testing the application. In ideal situation, the automated performance test solution should contain this monitoring Lack of integration is not an excuse for omitting this phase.

Server KPIs

Measuring the performance of server is done by monitoring software which observes the behavior of specific performance counters and metrics. In Unix/Linux world, these utilities are monitor, top, vmstat, iostat, SAR that monitors server KPIs. Monitorings using several layers is recommended, where the top layer is called generic monitoring, which focuses on a small amount of counters that will clearly tell if server is under stress. Next layers of monitoring should focus on specific technologies that are part of the application: such as the web, application and database servers.

There are several models or templates that can be used:

Generic templates is a common set of metrics that apply to all servers in the same tier having the same operation system. The purpose is to provide first-level monitoring of the effects of load and stress. These metrics are typically how busy the CPUs are and how much of memory is present.

These counters should be present in the template:

- Utilization percentage of processor
- Queue length of processor
- Available memory (preferably in bytes)
- Memory pages per second
- Measuring context switches (per time unit)
- Two for physical disk: average disk queue length and disk time

- Network interface: Packets Received and Outbound Errors

Database server tier

Most databases are similar in architecture, but differ from monitoring perspective. As a result, every database type will require its own template. Examples of database types:

- MySQL
- Oracle
- Microsoft SQL Server
- IBM DB2
- Sybase
- Informix

Network KPIs

While performance testing, packet round-trip time, data presentation and the detection of any errors that may occur as a result of high data volumes are the main focus of network monitoring. This capability might be built into automated performance test tool or it may be provided separately. However, if the guidelines on where to inject load were followed and the data presentation have been optimized, then network issues should be the least likely cause of problems during performance testing.

Available performance counters for Unix/Linux and Windows operating systems monitor the number of errors detected during a performance test execution as well as the amount of data being handled by each NIC card. Some automated performance test tools even separate server and network time for each element within a page to help differentiate between server and network problems.

3.4 Interpreting Results

It is vitally important to interpret the results of performance test correctly. Since proper performance target has been set as part of testing requirements, problems should be spotted quickly during the test or as part of the analysis at test completion. Another important thing to do is to have all the necessary information at hand for further diagnosis.

3.4.1 The Analysis Process

There are two approaches how to perform analysis: in real time (as the test executes) or at its conclusion.

Real-Time Analysis is basically waiting for something to happen or for the test to complete without apparent incident. When a problem occurs, monitoring tools are responsible for reporting the location of the problem in the application landscape.

3.4.2 Performance test output types

The root of performance test results analysis is statistical analysis. There are several types of statistical data that can be extracted from these results:

If **mean** [1] is mentioned, typically, arithmetic mean is implied. Mean is basically the average of a set of values. Response times are aggregated with mean to derive their average. Even though there exist many types of means, the most important for performance testing is 'arithmetic mean'. For example, the arithmetic mean of 3.5, 4.6, 4.0, 4.2 and 3.9 is 4.04. **Median** is the middle value in a set of numbers. It is essential for performance testing results interpretation.

Formal mathematical formula for the arithmetic mean:

$$\bar{x} = \frac{1}{n} \times \sum_{i=1}^{n} x_i$$

Standard deviation can be defined as variation or dispersion from the calculated average (mean) value. It is used to measure confidence in statistical conclusions. Data in random and real-life events tend to exhibit a **normal distribution** (also known as the Bell Curve 3.3).

Formal mathematical formula for the standard deviation:

$$\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{n - 1}}$$

Since high standard deviation most likely indicates erratic end user experience, an effort to achieve small standard deviation should be made. For example, when a mean response time is 60 seconds and standard deviation is 30 seconds, user has a high chance of experiencing response time from 45 seconds to 75 seconds.

(Note to 3.3: σ is symbol for standard deviation, μ stands for mean)

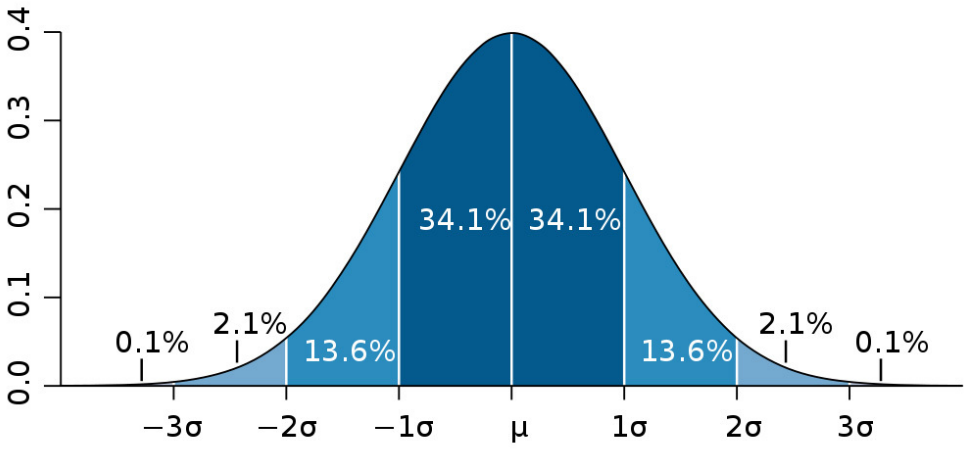


Figure 3.3: The Bell curve. Source [3]

Chapter 4

Performance test suite design

4.1 Real-life use cases and test cases

4.1.1 Use Cases

Use Case 1: Updating Case Runs

Mass CaseRun update	**UC01**
Description:	When working with Nitrate, significant part of the work is transferring the results of automatic tests to case run states. Updating several hundreds of case run states is really time consuming task, these updates can take tens of minutes. The impact is much greater since this scenario is quite common.
Culprit:	Updating status of multiple Case Runs is a very time consuming task when executed one-by-one (single update in one request)

Use Case 2: Print TestCase tags

Display Test Case Tags	**UC02**
Description:	Another common task is displaying tags of all test cases in a certain test run. It is extremely slow to filter test cases with specified tags in large test plans (hundreds of test cases). This task takes several minutes in that case, but there certainly is space for improvement.

Culprit:	The main problem is that tags class are not implemented. When all test cases are downloaded (in one call), they are converted one-by-one (thus causing significant slowdown) and communication with server is redundant.

Use Case 3: TestCase not cached in CaseRun

TestCase not cached in CR	**UC03**
Description:	Children test plan walkthrough of a master plan is rather ordinary task (especially when trying to gather information about case run states of all test runs in children test plans). It consumes a lot of time and there is space for improvement.
Culprit:	Test Cases linked to case runs are fetched every time from server (no use of cache)

Use Case 4: Search for test case and its owners

Test case and its owners	**UC04**
Description:	When trying to display all test cases of a test plan, usually users want to display even test case owner. It is used to contact person who is responsible for this test case in order to provide additional information (review) to test case.
Culprit:	The main problem is fetching users from database one by one.

Use Case 5: Search TestCase plans

Display TestCase plans	**UC05**
Description:	The last use case consists of regular display of test case along with every test plan that contains current test case. This is also a standard operation that python-nitrate users experience.
Culprit:	Test plans are downloaded twice (once in test case and then in test plan)

4.1.2 Test Cases

Test Case 1: Updating Case Runs

Covering mass CaseRun update use case is very important, because this is quite common task. The main problem is in the communication - every update is executed in a single call, instead of executing multiple updates in a single call. This feature is going to be very helpful especially for TestRuns with large number of TestCases.

Code:

```
for caserun in TestRun(self.performance.testrun):
    log.info('{0} {1}'.format(caserun.id, caserun.status))
    caserun.status = Status(random.randint(1,8))
    caserun.update()
```

Basic description:

First of all, TestRun is fetched from server along with all its case runs. For every case run, its state is changed to random state (because update happens only when state is changed) and perform update of a single case run.

Proposed solution:

The solution might be MultiCall feature.

Test Case 2: Print TestCase tags

Use case 2 is base for this test case. Showing tags of multiple test cases in a test plan is really complicated. The main problem is that tags class are not implemented. When all test cases are downloaded (in one call), they are converted one-by-one (thus causing significant slowdown).

Code:

```
for case in TestPlan(self.performance.testplan):
    log.info('{0}: {1}'.format(case, case.tags))
```

Basic description:

After initial fetch of test plan, it iterates through all of the test cases and gets id of test case's tag. This tag id is then converted into name using server call. This is very redundant, because one tag is fetched several times (more significant when a lot of test cases with the same tag).

Proposed solution:

This test case can be improved by implementing Tag class with caching.

Test Case 3: TestCase not cached in CaseRun

This test case shows the problem of handling test cases that occur frequently – for example linked to case runs. These case runs are linked to test runs and they are linked to test plans. Test runs within the same test plan are more or less similar, so caching test cases is really a good idea.

Code:

```
for testplan in TestPlan(self.performance.testplan).children:
    log.info('{0}'.format(testplan.name))
    for testrun in testplan.testruns:
        log.info('  {0} {1} {2}'.format(
                testrun, testrun.manager, testrun.status))
        for caserun in testrun.caseruns:
            log.info('    {0} {1} {2}'.format(
                    caserun, caserun.testcase, caserun.status))
```

Basic description:

The first step is fetching children of specified test plan. In every child plan, test runs are iterated and in every test run, case runs are displayed. For every case run, their test case is called and it is displayed, along with caserun status.

Proposed solution:

Solution can be implementing common caching class, so cached testcases can be used to improve performance.

Test Case 4: Display test case and its owner

Code:

```
for testcase in TestPlan(self.performance.testplan):
    log.info('{0}: {1}'.format(testcase.tester, testcase))
```

Basic description:

In the beginning, test cases of a specified test plan are fetched in a single call. Every test case has its tester and when the user is not initialized, user object is fetched from database. This repeats until every test case is displayed with its tester.

Proposed solution:

One solution is persistent local cache with stored objects (users, ...).

Test Case 5: Show all test plans linked to TestCase

This covers use case when displaying test plans with their test cases and all test plans of test cases.

Problem is with fetching the same test plan more than once when multiple test cases contain the same test plan.

Code:

```
for testcase in TestPlan(self.performance.testplan):
    log.info('{0} is in test plans:'.format(testcase))
    for testplan in testcase.testplans:
        log.info('  {0}'.format(testplan.name))
```

Basic description:

Initially, all testcases in a test plan are fetched from database and then is every test case processed one by one. Every test case fetches a list of test plans and test plans are fetched one by one.

Proposed solution:

Solution might be caching test plans (so only one fetch is required) or persistent cache.

Chapter 5

python-nitrate performance improvements

5.1 Feature enhancements

5.1.1 FE00: Test Suite

Performance test suite is the first step to application performance testing. During initial state of thesis, feedback from python-nitrate users was required and a lot of comments were received regarding performance issues. After processing these issues, five use cases that have to be improved in the terms of performance were created. These use cases are later converted into test cases, which performance test suite contains. Test cases can be found in the self test section of every affected class. The performance test suite can be run as:

```
# python api.py --performance [class]
```

Every performance test in test suite displays elapsed time in human readable format and the result of the test.

For running the performance test suite an additional section containing information about the test bed is required:

```
[performance]
testplan = 1234
testrun = 12345
```

test-bed-prepare.py script

Use the test-bed-prepare.py script attached in the test directory to prepare the structure of test plans, test runs and test cases. Nitrate test case management system instance is required to run this performance test suite.

This script creates tree structure consisting of a master plan, user specified number of test cases, test plans and test runs. These test cases are linked to test plans and also contain a random tag. Test runs are created in the end from test plans.

Usage of test-bed-prepare.py script:

```
    test-bed-prepare [--plans #] [--runs #] [--cases #]

Options:
    --plans=#   create specified number of plans
    --runs=#    create specified number of runs
    --cases=#   create specified number of cases
```

5.1.2 FE01: MultiCall

MultiCall [10] feature is used to encapsulate multiple calls to a remote server into a single request. If enabled, TestPlan, TestRun, TestCase and CaseRun objects will use MultiCall for updating their states (thus speeding up the process). Example usage:

```
    multicall_start()
        for caserun in TestRun(12345):
            caserun.status = Status(,,IDLE‘‘)
            caserun.update()
    multicall_end()
```

When multicall_start() is called, update queries are not sent immediately to server. Instead, they are queued and after multicall_end() is called, all queries are sent to server in a batch.

This feature enhancement resolves use case UC01, where CaseRun statuses are updated in one call instead of one call per one CaseRun update.

5.1.3 FE02: Tag Class

For handling tags of Test Cases, Test Runs and Test Plans, new class Tag is implemented. Tags are handled as objects instead being handled as strings, which provides much easier access and modification.

This is very important after implementing tags caching, because tags are stored in a single cache. This means when tag is fetched from server, it is immediately stored in Tag cache.

5.1.4 FE03: Common Caching

In order to save calls to server and time, caching support has been extended. Originally, caching support was present in *Category*, *Component*, *User* and *PlanType*. Now, every class that handles objects has its own cache and it is used to save already initialized (fetched) objects from server. Several classes are automatically fetched from server after initialization (immutable objects), the rest will be fetched from server upon request.

Currently, there are 4 types (levels) of caching:

```
CACHE_NONE - no caching at all
CACHE_CHANGES - caching only local updates of instance attributes
CACHE_OBJECTS - caching objects for further use (default setting)
CACHE_PERSISTENT - persistent caching (caching in a file) option enabled
```

There is a difference from original implementation of caching. The old version had CACHE_ALL caching type (fetch every object in a class from server), but it was implemented only in *User* class. Now, this type of caching is deprecated and it has been replaced with new type, **CACHE_PERSISTENT**.

Cache implementation has been improved in another way: the original cache implementation had separate `__new__()` function [7] in classes that supported caching, now it is implemented in *Nitrate* class, thus reducing 4 implementations to 1 unified implementation for caching in the whole module. Searching in cache is implemented in `_cache_lookup()` function that is defined in *Nitrate*, but has multiple redefinitions in subclasses that require special ways of searching.

5.1.5 FE04: Persistent cache

Persistent cache (local proxy) was another idea how to speed up performance of the module. It allows class caches to be stored in a file, load caches from a file, and clear caches. This performance improvement is very helpful mainly for immutable classes (for example User), where all user can be imported in the beginning of a script and a lot of connections can be saved.

This performance improvement can be activated only by specifying file name in config section ([cache]).

Cached objects expiration

Since there are many types of objects / classes in python-nitrate module, every class has to have expiration. This means that after certain period of time, instance of class has to be fetched again from the server in order to stay up-to-date. Expiration times of classes differ, for example, immutable classes have much longer expirations than mutable objects.

Cache expiration is a way how to prevent using probably obsoleted object (for example caserun). Every class has its own default expiration time, but, of course, it can be modified from config file (see example in [expiration]) and user input. Time unit in cache expiration is 1 second.

There are two special values:

```
NEVER_CACHE -> no caching of certain class (0 seconds)
NEVER_EXPIRE -> object never expires (100 years)
```

Table 5.1 shows expiration times in all classes:

Classes	Expiration
Normal	1 month
TestCase	1 hour
TestPlan	1 hour
TestRun	1 hour
CaseRun	0 hours

Table 5.1: Expiration times in persistent cache

5.1.6 FE05: Container initialization

Several containers support direct initialization of values. This saves a server call, so the performance is improved. This type of initialization is currently available only in *CaseTags*, *RunTags* and *PlanTags* classes. The principle is quite simple. When an input set is provided, values from the input are used instead of values from server. This direct initialization is currently enabled only if **CACHE_PERSISTENT** caching type is enabled.

5.2 Implementation and testing

All features listed in future features have been implemented into python-nitrate module in order to increase performance of this module. Development was incremental, that means after implementation of performance test suite and its integration, along with test-bed-prepare.py script, which creates needed structure in Nitrate instance, the focus was transferred to feature enhancements. Coding conventions in PEP 8 [6] Python style guide were followed in feature enhancement implementation.

First enhancement, MultiCall, was implemented in the beggining and the initial results were more than satisfactory. The initial speedup was around 200%, so that was considered as quite a success.

Tag class also boosted the performance. Tags are now treated as objects instead of strings or ids, so the access to tags is now unified (can be accessed both through ID or name). Since it is now a class, caching can be implemented for further performance improvement.

Common caching class is the biggest step in performance improvements, but probably has low impact on performance and is a ground for persistent caching and container initialization, which significantly improves performance.

Persistent cache is implemented to provide cache saving and loading, which makes the class objects accessible after python-nitrate exits and starts again. This feature has the potential to remarkably speed up the module in specific cases.

Container initialization was intended mainly to initialize values from input set of values in containers related to tags.

Performance testing of improvements is performed on server with Nitrate instance and three test plans are created. These test plans contain different number of test cases: 10, 100 and 1000 test cases. Every test plan has its child plan, which is basically the same test plan. One test run is created for every test plan.

The results of performance testing improvements are visible in Appendix A, where tables with times of all test cases are present.

5.3 Results

5.3.1 10 Test cases

	Test Case				
Type	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
Original implementation	32.26s	23.69s	16.83s	12.39s	44.80s
Improved implementation	13.57s	24.18s	42.24s	0.00s	101.98s
Improved + persistent cache	14.47s	5.57s	19.65s	0.00s	62.99s

Table 5.2: Comparison of results with 10 Test cases (mean)

	Test Case				
Type	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
Original implementation	3.39s	0.35s	0.36s	0.26s	0.96s
Improved implementation	1.47s	1.31s	1.99s	0.00s	6.89s
Improved + persistent cache	2.16s	1.51s	12.60s	0.00s	19.66s

Table 5.3: Comparison of results with 10 Test cases (standard deviation)

The obvious thing here is that even though there is great progress in first and fourth test case, regress occured in test case three and five in improved implementation. This is because all immutable objects are immediately fetched and cached, whereas in the old implementation it was only initialization (no contact with server). Plus, when set of objects was being fetched from server in original implementation, only one call was required. Second test case made progress only when persistent cache was enabled – otherwise the performance was comparable to old implementation.

5.3.2 100 Test cases

	Test Case				
Type	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
Original implementation	382.33s	204.05s	26.73s	26.64s	480.65s
Improved implementation	169.21s	248.77s	69.31s	0.00s	731.58s
Improved + persistent cache	152.69s	14.06s	18.61s	0.00s	514.02s

Table 5.4: Comparison of results with 100 Test cases (mean)

MultiCall feature again increased the performance of the module. The difference in tags initialization, that is enabled in CACHE_PERSISTENT caching mode, has now remarkably increased speed in second test case. Checking CaseRuns in TestRuns in TestPlans (TC03) took longer as expected, as well as fifth test case. Displaying test cases with their testers (TC04) uses cached information (no network activity).

	Test Case				
Type	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
Original implementation	12.41s	10.97s	1.34s	1.02s	15.95s
Improved implementation	11.71s	36.40s	7.88s	0.00s	101.00s
Improved + persistent cache	16.88s	2.27s	33.92s	0.00s	129.02s

Table 5.5: Comparison of results with 100 Test cases (standard deviation)

5.3.3 1000 Test cases

	Test Case				
Type	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
Original implementation	3596.04s	2578.52s	97.50s	148.44s	6088.95s
Improved implementation	1021.09s	2636.77s	262.06s	0.00s	6278.45s
Improved + persistent cache	996.48s	53.99s	164.63s	0.00s	6419.04s

Table 5.6: Comparison of results with 1000 Test cases (mean)

	Test Case				
Type	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
Original implementation	121.30s	6.63s	4.68s	5.36s	45.75s
Improved implementation	40.32s	49.71s	17.21s	0.00s	57.41s
Improved + persistent cache	11.71s	36.40s	7.88s	0.00s	101.00s

Table 5.7: Comparison of results with 1000 Test cases (standard deviation)

Huge difference is observed while displaying tags of test cases. That test case is nearly 48 times faster with Tag class implemented compared to original implementation. Multicall speeds up first test case 3.6 times, which is comparable to previous tests. Again, displaying test cases and their users does not take any time since all information is already cached. Remarkable deterioration is present in test cases 3 and 5 (reasons mentioned earlier).

5.3.4 Summary

To sum up, MultiCall improvement made a big difference in performance along with Tag class implementation. Caching is certainly responsible for speeding up specific test cases in the module. On the other hand, changes in initialization of classes and handling of objects significantly slowed down particular test cases. In the third test case with persistent cache enabled, first iteration takes much more time than the others (because of fetching objects from database). Objects cached in persistent cache are used in further iterations. Standard deviation is very high under these conditions.

Chapter 6

Conclusion

The goal of this thesis has been improvement of performance in python-nitrate module and implementation of caching support for all relevant classes. Measuring the impact of improvements required performance test suite, which was created in the beginning. Test cases present in this test suite were derived from use cases, that required large data manipulation and they were provided by python-nitrate users.

The main achievements are implementation of caching in a parent *Nitrate* class in this module, but also Persistent cache and MultiCall feature, which improved performance of updating part of module. Tag class is also introduced in order to provide better tag handling. Draft of container initialization is presented in Container classes handling tags, which also remarkably improves performance. Lastly, *test-bed-prepare.py* script has been implemented and it creates desired structure of test plans, test cases and test runs in Nitrate instance for running performance tests in test suite.

The important results are those created in the testing part that these feature enhancement really improved performance in python-nitrate module. We can see that several feature enhancements have large impact on performance, but some changes, unfortunately, make the module slower. More detailed commentary is present in result section.

Future of this thesis is bright, since feature enhancements created while working on this thesis are ground for further improvements and will be almost certainly part of upstream. There is still space for a lot of things in the future, for example creating container caching for all containers, rewriting Bugs class, fixing minor bugs, ...

Bibliography

[1] B.L. Agarwal. *Basic Statistics*. New Age International Pvt Ltd Publishers, 1st edition, 2009.

[2] Scott Barber. *Get performance requirements right – think like a user* [online]. http://www.perftestplus.com/resources/requirements_with_compuware.pdf, 2007 [cit. 2013-05-14].

[3] David L. Chandler. *Explained: Sigma* [online]. http://web.mit.edu/newsoffice/2012/explained-sigma-0209.html, 2012 [cit. 2013-05-14].

[4] David Malcolm, Yuguang Wang, June Zhang and Xuqing Kuang. *Nitrate: Test Case Management System* [online]. https://fedoraproject.org/wiki/Nitrate, 2012-03-15 [cit. 2013-05-04].

[5] G.L.Martin, K.G.Corl. System response time effects on user productivity. *Behaviour and Information Technology*, vol. 15 (no. 1):3–13, 1986.

[6] Guido van Rossum, Barry Warsaw. *PEP 8 – Style Guide for Python Code* [online]. http://www.python.org/dev/peps/pep-0008/, 2001 [cit. 2013-05-13].

[7] Mark Lutz. *Learning Python*. O'Reilly Media, 3rd edition, 2007.

[8] Ian Molyneaux. *The Art of Application Performance Testing*. O'Reilly Media, Cambridge, 1st edition, 2009.

[9] Petr Šplíchal, Zbyšek Mráz, Martin Kyral and Lukáš Zachar. *python-nitrate: Python API for the Nitrate test case management system* [online]. http://psss.fedorapeople.org/python-nitrate/, 2012 [cit. 2013-05-12].

[10] Python Software Foundation. *xmlrpclib – XML-RPC client access* [online]. http://docs.python.org/2/library/xmlrpclib.html, 2013 [cit. 2013-05-12].

[11] Python Software Foundation. *ConfigParser – Configuration file parser* [online]. http://docs.python.org/2/library/configparser.html, 2013 [cit. 2013-05-13].

Appendix A

Performance values

Short name	Long name
TC01	Updating multiple CaseRuns from a TestRun
TC02	Checking tags of test cases
TC03	Checking CaseRuns in TestRuns in TestPlans
TC04	Checking test cases and their default testers
TC05	Checking test plans linked to test cases
Mean	Arithmetic mean
SD	Standard deviation

Table A.1: Test Case names

10TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	31.38s	23.48s	14.94s	13.44s	45.24s
2.	32.66s	23.54s	16.77s	12.26s	44.91s
3.	29.84s	23.59s	17.15s	12.18s	44.77s
4.	35.41s	23.40s	16.73s	12.26s	44.72s
5.	38.27s	23.51s	16.80s	12.26s	43.84s
6.	35.53s	25.14s	16.67s	12.73s	44.39s
7.	32.82s	23.91s	16.96s	12.90s	44.66s
8.	35.48s	23.53s	17.05s	12.64s	44.59s
9.	35.58s	23.47s	16.94s	12.30s	45.43s
10.	29.79s	23.49s	16.73s	12.27s	44.39s
11.	32.59s	23.58s	16.78s	12.26s	44.65s
12.	30.40s	23.52s	16.76s	12.36s	45.19s
13.	30.13s	23.98s	16.77s	12.43s	44.85s
14.	24.37s	23.72s	16.76s	12.73s	44.57s
15.	27.20s	24.08s	17.79s	12.35s	45.17s
16.	29.79s	23.96s	16.81s	12.48s	44.90s
17.	29.91s	23.44s	16.80s	12.32s	44.72s
18.	35.50s	23.51s	16.76s	12.28s	44.97s
19.	36.10s	23.46s	16.95s	12.26s	44.44s
20.	27.48s	23.41s	16.79s	12.27s	43.88s
21.	35.49s	23.69s	16.80s	12.45s	44.59s
22.	32.60s	23.49s	17.00s	12.17s	44.25s
23.	35.55s	23.54s	16.69s	12.23s	45.46s
24.	29.71s	23.48s	16.80s	12.30s	44.36s
25.	29.77s	23.84s	16.84s	12.48s	45.10s
26.	35.88s	23.52s	16.80s	12.19s	44.61s
27.	35.47s	23.63s	16.88s	12.27s	43.96s
28.	32.81s	24.10s	16.75s	12.23s	44.19s
29.	35.43s	23.44s	16.79s	12.33s	44.71s
30.	35.55s	23.51s	16.78s	12.26s	44.60s
31.	35.55s	23.44s	17.12s	12.25s	44.71s
32.	26.80s	23.44s	16.76s	12.32s	43.85s
33.	26.99s	23.51s	17.19s	12.26s	50.15s
34.	33.05s	24.63s	16.92s	13.01s	44.69s
35.	30.03s	23.66s	16.71s	12.24s	44.34s
36.	32.53s	23.57s	16.81s	12.28s	44.27s
37.	27.01s	23.71s	16.85s	12.22s	45.14s
38.	35.44s	23.58s	16.74s	12.28s	44.42s
39.	35.61s	23.95s	16.91s	12.78s	44.98s
40.	33.17s	24.14s	17.03s	12.29s	45.58s
Mean	32.36s	23.69s	16.83s	12.39s	44.80s
SD	3.39s	0.35s	0.36s	0.26s	0.96s

Table A.2: Results of run with original implementation (10 Test cases)

10TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	15.17s	23.90s	42.23s	0.00s	112.40s
2.	13.02s	25.19s	42.26s	0.00s	121.19s
3.	14.13s	24.31s	42.06s	0.00s	117.27s
4.	14.64s	29.19s	42.90s	0.00s	98.272s
5.	13.98s	23.68s	41.52s	0.00s	101.67s
6.	14.95s	23.43s	41.51s	0.00s	98.521s
7.	18.64s	24.16s	41.52s	0.00s	97.526s
8.	11.44s	23.68s	41.42s	0.00s	97.701s
9.	12.97s	23.61s	41.57s	0.00s	97.988s
10.	14.12s	23.61s	41.64s	0.00s	97.747s
11.	14.96s	23.48s	41.62s	0.00s	99.564s
12.	13.03s	23.91s	41.45s	0.00s	100.25s
13.	14.11s	23.60s	42.60s	0.00s	97.842s
14.	13.22s	24.96s	42.49s	0.00s	99.893s
15.	11.48s	24.37s	42.16s	0.00s	99.963s
16.	14.16s	23.72s	41.89s	0.00s	98.834s
17.	15.10s	25.35s	43.10s	0.00s	105.80s
18.	10.53s	24.38s	44.62s	0.00s	116.31s
19.	14.94s	23.67s	42.41s	0.00s	114.44s
20.	14.01s	23.64s	41.48s	0.00s	114.85s
21.	14.08s	23.82s	42.85s	0.00s	111.51s
22.	13.03s	23.41s	42.77s	0.00s	111.25s
23.	11.34s	23.54s	41.36s	0.00s	100.05s
24.	13.03s	23.48s	41.17s	0.00s	98.225s
25.	13.04s	26.47s	42.55s	0.00s	97.650s
26.	14.04s	23.85s	42.15s	0.00s	96.986s
27.	13.42s	29.17s	53.79s	0.00s	97.544s
28.	14.02s	23.66s	41.43s	0.00s	98.086s
29.	14.98s	23.46s	41.51s	0.00s	97.438s
30.	11.38s	23.89s	41.48s	0.00s	97.184s
31.	11.38s	23.79s	41.49s	0.00s	97.539s
32.	14.05s	23.53s	42.05s	0.00s	102.49s
33.	13.07s	24.35s	41.34s	0.00s	98.271s
34.	11.45s	23.56s	41.44s	0.00s	97.616s
35.	13.93s	23.51s	41.41s	0.00s	97.681s
36.	15.01s	23.40s	41.58s	0.00s	97.101s
37.	14.07s	23.63s	42.76s	0.00s	99.172s
38.	13.08s	23.58s	41.34s	0.00s	97.714s
39.	13.08s	23.86s	41.41s	0.00s	97.779s
40.	12.94s	23.60s	41.61s	0.00s	100.17s
Mean	13.57s	24.18s	42.24s	0.00s	101.98s
SD	1.47s	1.31s	1.99s	0.00s	6.89s

Table A.3: Results of run with improved implementation (10 Test cases)

10TCs	**Test Case**				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	10.42s	4.67s	83.27s	0.00s	130.48s
2.	13.97s	4.76s	15.09s	0.00s	50.13s
3.	14.01s	4.74s	15.07s	0.00s	50.15s
4.	14.02s	4.74s	15.09s	0.00s	51.21s
5.	11.41s	4.72s	15.16s	0.00s	50.63s
6.	13.03s	4.68s	15.19s	0.00s	51.97s
7.	13.04s	4.67s	15.15s	0.00s	50.58s
8.	13.18s	4.74s	15.58s	0.00s	50.87s
9.	13.31s	7.71s	15.19s	0.00s	86.07s
10.	18.12s	8.31s	22.08s	0.00s	93.20s
11.	14.10s	4.78s	17.79s	0.00s	77.89s
12.	19.75s	8.52s	25.72s	0.00s	94.78s
13.	18.15s	8.66s	24.65s	0.00s	87.71s
14.	19.04s	8.54s	25.22s	0.00s	79.28s
15.	17.74s	8.46s	23.78s	0.00s	66.43s
16.	14.05s	4.77s	15.07s	0.00s	77.80s
17.	15.00s	4.74s	20.05s	0.00s	50.38s
18.	15.00s	4.73s	15.05s	0.00s	50.74s
19.	14.20s	4.73s	15.10s	0.00s	50.58s
20.	13.01s	4.72s	15.11s	0.00s	50.65s
21.	13.08s	5.11s	15.42s	0.00s	50.83s
22.	15.01s	4.77s	15.28s	0.00s	50.58s
23.	14.94s	4.72s	15.18s	0.00s	52.00s
24.	13.06s	4.74s	15.18s	0.00s	50.55s
25.	12.12s	4.77s	15.22s	0.00s	68.24s
26.	14.49s	7.48s	25.75s	0.00s	51.51s
27.	15.04s	4.96s	15.40s	0.00s	63.63s
28.	15.00s	4.75s	16.98s	0.00s	50.02s
29.	13.02s	4.66s	15.40s	0.00s	50.36s
30.	13.99s	4.76s	15.28s	0.00s	50.68s
Mean	14.47s	5.57s	19.65s	0.00s	62.99s
SD	2.16s	1.51s	12.60s	0.00s	19.66s

Table A.4: Results of run with improved implementation and persistent cache (10 Test cases)

100TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	387.32s	203.47s	26.92s	26.36s	537.17s
2.	410.50s	245.04s	32.15s	27.24s	488.32s
3.	381.52s	197.35s	26.20s	26.27s	476.62s
4.	375.46s	200.28s	26.56s	26.36s	473.17s
5.	377.97s	199.44s	26.11s	25.88s	469.89s
6.	385.58s	199.90s	26.43s	26.71s	473.94s
7.	366.94s	200.92s	26.29s	26.21s	473.82s
8.	370.27s	220.55s	26.27s	30.19s	510.02s
9.	384.30s	200.16s	27.44s	26.57s	477.51s
10.	379.99s	204.30s	26.08s	26.83s	483.67s
11.	397.09s	201.62s	26.37s	26.35s	479.87s
12.	389.61s	200.55s	27.32s	26.03s	472.05s
13.	393.08s	199.58s	26.43s	26.30s	475.68s
14.	367.32s	199.48s	26.11s	28.58s	478.47s
15.	373.31s	198.76s	26.72s	26.03s	471.10s
16.	391.41s	200.39s	25.96s	26.12s	472.16s
17.	399.34s	199.52s	26.42s	26.11s	479.01s
18.	384.74s	210.97s	26.81s	26.17s	472.88s
19.	365.13s	199.58s	26.04s	26.35s	474.90s
20.	365.73s	199.32s	25.97s	26.16s	472.93s
Mean	382.33s	204.05s	26.73s	26.64s	480.65s
SD	12.41s	10.97s	1.34s	1.02s	15.95s

Table A.5: Results of run with original implementation (100 Test cases)

100TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	162.32s	256.14s	58.91s	0.00s	602.19s
2.	159.77s	198.58s	61.80s	0.00s	603.15s
3.	185.12s	256.29s	71.12s	0.00s	929.54s
4.	205.02s	216.56s	66.04s	0.00s	817.48s
5.	162.86s	208.00s	59.53s	0.00s	618.50s
6.	156.96s	197.90s	58.64s	0.00s	606.39s
7.	153.63s	201.59s	59.21s	0.00s	602.34s
8.	160.70s	200.73s	59.07s	0.00s	624.24s
9.	155.45s	203.49s	58.75s	0.00s	601.45s
10.	168.48s	277.40s	76.88s	0.00s	859.51s
11.	172.30s	319.80s	75.37s	0.00s	829.94s
12.	167.94s	272.85s	78.62s	0.00s	777.54s
13.	178.71s	269.11s	75.97s	0.00s	777.58s
14.	170.68s	266.05s	74.25s	0.00s	771.62s
15.	177.66s	286.54s	76.24s	0.00s	781.20s
16.	164.28s	275.11s	76.90s	0.00s	768.95s
17.	167.28s	266.43s	75.23s	0.00s	762.79s
18.	173.85s	270.96s	75.76s	0.00s	777.96s
19.	172.86s	268.25s	73.71s	0.00s	759.46s
20.	168.46s	263.67s	74.28s	0.00s	759.89s
Mean	169.21s	248.77s	69.31s	0.00s	731.58s
SD	11.71s	36.40s	7.88s	0.00s	101.00s

Table A.6: Results of run with improved implementation (100 Test cases)

100TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	177.68s	16.54s	162.47s	0.00s	1055.53s
2.	141.29s	11.15s	9.57s	0.00s	477.67s
3.	164.43s	11.11s	8.87s	0.00s	509.63s
4.	169.58s	16.32s	9.65s	0.00s	455.70s
5.	162.82s	13.22s	14.6s	0.00s	500.52s
6.	151.65s	11.49s	14.8s	0.00s	504.28s
7.	123.22s	13.62s	8.39s	0.00s	507.50s
8.	140.22s	16.13s	8.25s	0.00s	480.63s
9.	133.40s	16.56s	9.33s	0.00s	468.15s
10.	169.79s	17.75s	13.69s	0.00s	507.21 s
11.	158.40s	13.63s	13.66s	0.00s	498.86 s
12.	137.23s	14.61s	10.97s	0.00s	486.95 s
13.	122.91s	11.31s	10.12s	0.00s	473.61 s
14.	173.35s	11.55s	9.44s	0.00s	464.37s
15.	163.54s	14.83s	9.74s	0.00s	459.02s
16.	150.15s	14.73s	11.13s	0.00s	442.35 s
17.	142.64s	16.12s	9.56s	0.00s	494.08s
18.	137.55s	12.64s	12.8s	0.00s	485.47s
19.	166.70s	11.11s	12.0s	0.00s	497.32s
20.	167.25s	16.84s	13.5s	0.00s	511.64s
Mean	152.69s	14.06s	18.61s	0.00s	514.02s
SD	16.88s	2.27s	33.92s	0.00s	129.02s

Table A.7: Results of run with improved implementation and persistent cache (100 Test cases)

100TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	3500.39s	2578.83s	102.95s	151.36s	6145.10s
2.	3775.46s	2582.46s	99.12s	151.21s	6034.16s
3.	3480.04s	2568.07s	91.62s	143.05s	6124.31s
4.	3654.24s	2577.65s	100.10s	142.39s	6082.74s
5.	3570.07s	2585.62s	93.75s	154.21s	6058.46s
Mean	3596.04s	2578.52s	97.50s	148.44s	6088.95s
SD	121.30s	6.63s	4.68s	5.36s	45.75s

Table A.8: Results of run with original implementation (1000 test cases)

1000TCs	Test Case				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	1066.07s	2599.66s	246.52s	0.00s	6214.93s
2.	975.58s	2642.39s	257.12s	0.00s	6327.70s
3.	1054.89s	2692.51s	284.32s	0.00s	6284.81s
4.	1023.37s	2574.27s	275.72s	0.00s	6340.08s
5.	985.58s	2675.06s	246.65s	0.00s	6224.74s
Mean	1021.09s	2636.77s	262.06s	0.00s	6278.45s
SD	40.32s	49.71s	17.21s	0.00s	57.41s

Table A.9: Results of run with improved implementation (1000 test cases)

1000TCs	**Test Case**				
#	**TC01**	**TC02**	**TC03**	**TC04**	**TC05**
1.	981.42s	51.39s	453.18s	0.00s	6663.83s
2.	965.57s	51.39s	97.55s	0.00s	6375.11s
3.	958.05s	51.23s	93.98s	0.00s	6525.33s
4.	1023.49s	50.52s	90.79s	0.00s	6306.42s
5.	1053.87s	65.43s	87.68s	0.00s	6224.55s
Mean	169.21s	248.77s	69.31s	0.00s	731.58s
SD	11.71s	36.40s	7.88s	0.00s	101.00s

Table A.10: Results of run with improved implementation and persistent cache (1000 test cases)

Appendix B

Contents of CD

The CD contains these folders:

- `python-nitrate` – repository with source codes of python-nitrate
- `thesis-tex` – source codes of this thesis
- `thesis-pdf` – this thesis in PDF format

Appendix C

Commits in python-nitrate git repository

```
Test bed prepare script
https://github.com/psss/python-nitrate/commit/796bbda

    Creates a master test plan with child test plans & runs
    Prepares a set of test cases with random tags & testers
    Links test cases to all test plans and test runs

Performance test cases
https://github.com/psss/python-nitrate/commit/9dcdb7b

    Configuration example
    Internal utility for printing time
    Initial set of performance test cases

Tag class implementation
https://github.com/psss/python-nitrate/commit/6bc6109

    New Tag class with support for caching implemented
    PlanTags, RunTags and CaseTags containers adjusted
    Relevant Tag test cases and create example updated

MultiCall support
https://github.com/psss/python-nitrate/commit/01cec90

    Implemented global functions for handling multicall mode
    Support in TestPlan, TestRun, TestCase and CaseRun update
    Added test case measuring CaseRun.status update performance

Common Caching
https://github.com/psss/python-nitrate/commit/93d5917

    Caching now handled in Nitrate instead of separate implementations
```

```
    New caching class methods: \_cache\_lookup() and \_is\_cached()
    Attribute initialization moved to a separate \_init() method
    Many adjustments in handling object fetching and initialization
    A bunch of new test cases covering the caching functionality

Persistent Cache
https://github.com/psss/python-nitrate/commit/760b042

    New Cache class handling persistence and expiration
    Various class adjustments to handle object expiration
    Cache level detection from the user config file
    Added detailed documentation for all cache features

Container Initialization
https://github.com/psss/python-nitrate/commit/6d69ac2

    Suport for direct initialization of container objects
    Implemented direct init for PlanTags, RunTags and CaseTags
```